SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

AD-A217 880

1b. RESTRICTIVE MARKINGS

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for public release; distribution unlimited.

5. MONITORING ORGANIZATION REPORT NUMBER(S)
AFOSR-TR- 90-0119

6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
Indiana University		Air Force Office of Scientific Research

6c. ADDRESS (City, State, and ZIP Code)	7b. ADDRESS (City, State, and ZIP Code)
Department of Computer Science Bloomington, IN 47405	Building 410 Bolling AFB, DC 20332-6448

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
AFOSR	NM	AFOSR-84-0372

8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
Building 410 Bolling AFB, DC 20332-6448	61102F	2304	A2	

11. TITLE (Include Security Classification)
Probabilistic Analysis of Algorithms for NP-complete Problems

12. PERSONAL AUTHOR(S)
John Franco

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
FINAL	FROM 30 Sep84 TO 29 Sep89		

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This is the final scientific report describing results obtained under Air Foce Office of Scientific Research grant number AFOSR-84-0372. The main objective was the probabilistic sense, it is easy to find a satisfying truth assignment to an instance of satisfiability but it is hard to verify that an unsatisfiable instance has a solution. A side issue was the analysis of probabilistic models used to obtain the main results.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Dr. Abraham Waksman	(202) 767-5027	NM

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

UNITED STATES AIR FORCE
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
BUILDING 410, BOLLING AFB, D.C. 20332

Grant No. AFOSR 84-0372

FINAL SCIENTIFIC REPORT

September 1984 to September 1989

Probabilistic Analysis of Algorithms for NP-complete Problems

John Franco, Principal Investigator
Department of Computer Science
Indiana University
Bloomington, Indiana 47405

ABSTRACT

This is the final scientific report describing results obtained under Air Force Office of Scientific Research grant number AFOSR 84-0372. The main objective was the probabilistic analysis of algorithms for the Satisfiability problem. The main results were that, in a probabilistic sense, it is easy to find a satisfying truth assignment to an instance of Satisfiability but it is hard to verify that an unsatisfiable instance has a solution. A side issue was the analysis of probabilistic models used to obtain the main results.

TABLE OF CONTENTS

1. Research Objective and Main Results

The goal of this research is to develop and analyse algorithms which can, in some practical sense, solve NP-complete problems quickly. NP-complete problems appear in many disciplines such as Cryptology, Operations Research, Artificial Intelligence and Computer System Design. NP-complete problems are the "hardest" of a class of problems known as NP. Associated with each NP problem we consider is an infinite set of instances. Instances may take the form of graphs, logic expressions, sets or many other structures depending on the problem. Each instance has a *size* denoted by n. Although the size of an instance I may be formally defined as the number of bits needed to efficiently encode I, for our purposes, we may regard the size of I to be the number of distinct objects in I. So, for example, a graph containing E edges and Q vertices has size $n = E + Q$. Associated with each instance I is a set of variables, a set of values that can be assigned to each variable and a constraint function U_I that maps value assignments to variables to $\{true, false\}$. For example, if I is a graph with Q vertices we might associate $Q - 1$ variables which take edge labels as values and a constraint function which has value *true* if and only if the edge set corresponding to the assignment given to the variables is a spanning tree of I. An assignment t such that $U_I(t) = true$ is a solution to I. An algorithm solves I if it determines whether or not a solution exists for I.

A problem in NP is said to be solved efficiently if there is an algorithm which solves every instance of the problem in time bounded by a polynomial in n. Unfortunately, there is no known computational scheme for efficiently solving any NP-complete problem and it is considered highly unlikely that one will be found (see [2] and [18]). Thus, every known method for solving an NP-complete problem P cannot find the solution to some instances of P in a reasonable amount of time. Furthermore, there is little hope that even an effective randomized algorithm (see [19], [27] and [28]) will be found for any NP-complete problem since, as is well known, this would imply an unlikely collapse of the polynomial hierarchy. However, if a method A can be found to efficiently find solutions to all but a few instances of P then A might be a practical method for solving P. We are interested in such (A, P) pairs.

We use probability theory to measure success in meeting our goal. A distribution D is assigned to the set of all possible instances of P of size n and we prove one of three kinds of results for a given algorithm A:

a) A finds a solution to an instance of P chosen randomly according to D in time bounded by a polynomial in n with probability greater than some positive constant κ as n gets large. Then we say A efficiently solves P *in bounded probability* under D.

b) A finds a solution to an instance of P chosen randomly according to D in time bounded by a polynomial in n with probability approaching 1 as n gets large. Then we say A efficiently solves P *in probability* under D.

c) A solves all of a large sample of instances of P chosen randomly according to D in average time that is bounded by a polynomial in n as n gets large. Then we say that A solves P *in polynomial average time.*

Results of type (a) are weaker than results of type (b) and results of type (b) are weaker than results of type (c). It is often the case that we can prove a weaker result but not a stronger one for a particular (A, P) pair under D. Although a type (c) result is the strongest type of result, even a type (b) result will allow us to conclude that A, in some practical sense (at least under D), efficiently solves P. A result of type (a) cannot always allow us to draw the same conclusion since κ may be very small (say .01). However, many algorithms that we consider make repeated attempts at finding a solution and, even if κ is small, there is a good chance that one will be found after several attempts as explained below.

Many algorithms we consider proceed by assigning values to variables in some order which is decided during computation and assignments are never undone either totally or partially. These algorithms either continue until all variables are assigned values (in which case a solution has been obtained) or they stop prematurely because they discover that every set of assignments of values to unassigned variables cannot possibly lead to a solution (in which case it cannot be determined whether or not a solution exists). A property of these algorithms is that the next variable to be assigned a value is chosen randomly from a large group of possibilities. Thus, repeated runs of such algorithms will execute differently and possibly give different results. If the probability that a run finds a solution is bounded from below by a constant and all runs execute independently then only a constant number of runs would be necessary for us to solve a random instance of P with probability arbitrarily close to 1 (this can be strengthened to a type (b) result if the number of runs is allowed to grow slightly with n). Unfortunately, it is not the case that all runs execute independently. However, for the algorithms we consider, the dependence is very weak and, according to the results of our experiments, we are justified in supposing that a small number of repeated runs of A will allow us to solve P with probability tending to 1. Thus, a result of type (a) seems to translate to a result of type (b) for the kinds of algorithms we consider. When referring to results of either type (a), (b) or (c) we will sometimes use the phrase *"probabilistically efficient"*.

Others have taken this approach for specific NP-complete problems. Algorithms which are probabilistically efficient have been found for the Hamiltonian Circuit problem [1] and

[23], the Planar Traveling Salesman problem [22], the Processor Scheduling problem [9], the Bin Packing problem [21] and other NP-complete problems. We have looked for similar results on algorithms for the Satisfiability problem.

An instance of the Satisfiability problem is, for the purposes of this research, a Boolean expression in Conjunctive Normal Form (CNF). The disjunctions, also called clauses, contain a subset of the set of positive and negative literals obtained from a set V of Boolean variables. An assignment of truth values to the variables of V is called a truth assignment to V. If $v \in V$ is assigned the value *true* then the positive literal v has the value *true* and the negative literal $\bar{v}$ has the value *false*. The two *unit* clauses (v) and $(\bar{v})$ are said to be complementary. A clause is satisfied if one or more of its literals has value *true*. An instance is satisfiable if there exists a truth assignment t to V which satisfies all clauses in it. Such an instance is said to be satisfied by t. The Satisfiability problem is, given an instance I, find a truth assignment which satisfies I, if one exists, or verify that no such truth assignment exists.

In order to understand performance over a range of instance types we attempt to get results for a family of distributions. We call such a family an input model or instance model.

Our main results, based on two input models, show the existence of probabilistically efficient algorithms for solving random instances of Satisfiability which are satisfiable with high probability. These results may be found in [5], [6], [7], [11], [12], [13], and [17]. The best algorithms are variants of the Davis-Putnam procedure which choose elimination variables successively and dynamically from clauses containing the least number of unassigned variables. The breakthrough in attaining these results is the application of flow analysis techniques in which clauses are regarded as objects which flow into and out of levels, where level i represents the set of clauses containing exactly i unassigned variables. A flow of less than one clause per iteration into the bottom level (one unassigned variable in a clause) can be handled without accumulation in the bottom level by choosing to assign a value which satisfies a clause at the bottom level. A flow greater than one per iteration results in an accumulation analogous to a bathtub overflowing because the drain is too small. Heavy accumulation at the bottom level increases the probability that two complementary clauses exist there. In such a case a satisfying truth assignment cannot be found. This mechanistic way to look at the operation of the algorithm has provided great insight into its probabilistic performance.

On the other hand, it is sometimes the case that a family of algorithms almost always requires exponential time to solve random instances of an NP-complete problem. For example, in [8] it is shown that a powerful formal system for determining the stability

number of a graph requires exponential time on almost all randomly generated graphs with a sufficiently large linear number of edges. We have obtained similar pessimistic results for Search Rearrangement Backtracking (a general, powerful family of search procedures which includes the Davis-Putnam Procedure [10]) on the Satisfiability problem when random instances are nearly always unsatisfiable [14]. Stated another way this result says that, regardless of what heuristic one uses, verification that a given instance is unsatisfiable will take exponential time on the average. An important aspect of this result is that the analysis explains why no heuristic can possibly be probabilistically efficient. The pessimistic result is due to two properties that most random instances have: each variable appears in at most a small fraction of the clauses, and the number of times that variables are linked in large enough subsets of the original set of clauses is not much greater than the number of clauses in the subset (if a variable v is in five clauses of a particular subset of clauses then four links are contributed by v to the number of links). In fact, any instance with these two properties is "hard" in the sense that no heuristic (on top of backtracking) can solve it in polynomial time.

Since probabilistic results of the kind stated above depend on input distribution, some justification and analysis of the input model is desirable. Part of our work has been to investigate properties of input models which induce probabilistic efficiency. We have found that some models generate a preponderance of "trivial" instances: those which can be solved by an algorithm that would be considered too weak to be used in practice. For example, some models allow clauses which contain no literals (null clauses). But instances which contain a null clause cannot be satisfied by any truth assignment to the variables contained in them. Thus, if a null clause exists in a random instance with probability tending to 1, then random instances are efficiently solved in probability simply by searching the input for a null clause. Clearly, this algorithm would be useless in practice.

However, some favorable results, on backtracking variants, which appear in the literature (e.g. [25], and [26]) depend on the high frequency of null clauses generated by the input model, although this fact is hidden in the analysis. We have shown ([12], and [13]) that even exhaustive search, after checking the input for a null clause and finding none, is average-case superior to the algorithms analyzed in the citations above because it usually stops early due to a null clause. More remarkably, exhaustive search was shown by us [13] to run in polynomial average time under the same conditions that the relatively sophisticated pure-literal-rule algorithm was shown to require superpolynomial average time [3] (because this algorithm failed to check for null clauses first). Thus, our investigations into input model properties have provided great insights into the nature of previous results and the utility of past and future results in this area.

The remainder of this report details the results we have attained under the grant.

In addition to the work on Satisfiability we have collaborated with other researchers on related problems in Quadtree representations (parallel architectures), hashing with lasy deletions (data base theory), and VLSI testing and verification. Sections 2 and 3 describe our results on algorithms for Satisfiability. Sections 4 to 6 describe results in the other areas. Sections 7,8, and 9 list publications under the grant, recent invited talks, and recent professional service.

3. Overview of Results for the Satisfiability Problem

The Satisfiability problem (SAT) is the first problem found to be NP-complete. Any problem in NP can easily be transformed to SAT and transformations from SAT to other NP-complete problems are often straightforward. SAT is, therefore, one of the more important NP-complete problems.

SAT is also an important problem because it turns up in a number of practical areas. For example, a collection of propositions P and a hypothesis H can be transformed to an instance I_1 of SAT such that I_1 is not satisfiable if and only if H follows logically from P. Thus, SAT is the basis for a number of theorem provers and is of interest to the Artificial Intelligence community. SAT also appears in automatic hardware testing and design as the following examples show:

1. A combinational circuit C computes one Boolean function of its inputs for each of its outputs. Its functionality can, therefore, be described by a set of Boolean formulae $\{B_1(C), B_2(C), ..., B_m(C)\}$, one for each output. A test sequence S for C must set input values to exercise both logic levels of each output of C. Finding a set of input values which forces the i^{th} output to level 1 (0) is equivalent to finding a truth assignment to the variables of $B_i(C)$ which satisfies $B_i(C)$ ($\neg B_i(C)$). Thus, the problem of generating (designing) a test sequence for a combinational circuit that checks for stuck-at faults can be stated as an instance of SAT (actually a set of instances of SAT).

2. The VLSI design process typically proceeds through many levels of abstraction from the functional level through the gate level down to the layout level. Functional equivalence must be maintained after each translation from one level to the next level down or else the end product may not perform as expected. If the circuit is combinational, functional equivalence may be regarded as Boolean equivalence between two levels of abstraction. Without going into the details of how one can obtain a Boolean formula describing the functionality of a particular level of abstraction, one way to test for Boolean equivalence at different levels is to determine whether the Boolean formula formed from the exclusive-or of the formulas at both levels is a tautology: if it is then the circuit descriptions at both levels are identical, otherwise they are not. But the problem of determining whether B is a tautology is equivalent to the problem of determining whether $\neg B$ has no solutions (this is an instance of SAT). Thus, SAT is important to functional verification (also known as logic verification) between different levels of abstraction in the design of VLSI combinational circuits. The problem of testing for functional equivalence between different levels of abstraction in the design

of sequential circuits may be reduced to a combinational circuit problem by means of the level sensitive scan design or equivalent approaches. Thus, SAT is important to VLSI design over a wide variety of circuit types.

Unfortunately, SAT is an NP-complete problem. Therefore, there is no known efficient algorithm for SAT (an algorithm that has running time bounded by a polynomial on the length of the given instance is considered efficient). Thus, the best that can be hoped for is an algorithm, based on some heuristic, which requires polynomial time on most instances. If the probability that an algorithm A for SAT runs in polynomial time tends to 1 as instance size increases then A may be regarded as an efficient algorithm for SAT in a practical (or at least probabilistic) sense. We have been looking at the question of whether probabilistically efficient algorithms exist for SAT and other NP-complete problems.

In order to answer the question of probabilistically efficient algorithms for SAT we must impose some distribution on instances. This presents two problems regarding the robustness of probabilistic results. First, a result obtained under one distribution does not necessarily hold under another (or, more appropriately, analyzed behavior assuming one distribution may be dramatically different from empirical behavior on a naturally occurring set of instances). Second, in order to produce an analysis at all it is practically a requirement that all "components" of a given instance be independent. In our work instances are CNF Boolean expressions and "components" are clauses. We use two models for constructing random instances. In both models a random instance of SAT consists of n clauses, each containing literals from r variables. In model M_1 each clause contains exactly k literals and is chosen uniformly from the set of all possible k literal clauses. In model M_2 each clause contains each literal with probability p (so clauses may have any number of literals up to $2r$). In order to reduce the effect of the two problems mentioned above we allow r and p to be functions of n; this allows us to adjust the properties of random instances to closely match the properties of many natural sets of instances. For example, consider how we might set the parameters of model M_2 to generate instances with properties that match instances of the test design problem of item (4) above (we omit a similar discussion of M_1). Suppose $p(n) = \alpha \ln(n)/r(n)$, $.4 < \alpha < 1$ (this restriction is not really necessary but it gives a useful example). Then random instances are usually satisfiable if, for any $\epsilon > 0$, $\lim_{n,r\to\infty} n^{1-\alpha}/r^{1-\epsilon} < \infty$, and instances are usually unsatisfiable if $\lim_{n,r\to\infty} n^{1-\alpha}/r = \infty$ (the higher or lower the rate of growth of $n^{1-\alpha}/r$, the more unsatisfiable or satisfiable, respectively, random instances are). Since all interesting combinational circuits correspond to Boolean formulas that are satisfiable, inputs to the test design problem of item (1) above have the property that they are satisfiable. Hence, we should make the function $f(n) = n^{1-\alpha}/r(n)$ tend rapidly to zero to generate random instances that closely match

this property. It turns out that it probably does not matter exactly how fast $f(n)$ tends to zero because one of our main results is that SAT is a "probabilistically easy" problem if $f(n)$ tends to zero, regardless of how fast $f(n)$ tends to zero (we caution the reader not to make too much of the relationship between M_2 and instances of test design at this time; the preceeding example merely illustrates a general connection between the two).

Our main results, in general terms, are as follows:

a. Almost all satisfiable instances of SAT generated under M_2 can be solved in $O(n \ln(n))$ time.

b. Almost all "interesting" unsatisfiable instances of SAT generated under M_1 require exponential time to solve (that is, verify unsatisfiability) by Backtracking using any heuristic for variable elimination.

The results are stated precisely in Section 3. We should like to point out that "Backtracking using any heuristic" represents a wide class of algorithms so the result of item (b) is fairly strong. The results suggest that SAT is "probabilistically easy" if most inputs are satisfiable and is "probabilistically hard" if most inputs are unsatisfiable. Translating this to the Design Automation problems raised earlier, the results suggest that the test design problem is "probabilistically easy" but the functional or logic verification problem is "probabilistically hard". Actually, this phenomenon has been observed in the Design Automation community for some time. However, the meaning of "easy" and "hard" has not been understood. Experiments have shown that some algorithms are faster than others for certain classes of inputs of a certain size (for example, see [20], [24] and [29]) but these experiments do not seem to say how fast, in an absolute sense, over many classes of inputs (even unforseen), and many different sizes. Also, these experiments usually do not give insight as to why the proposed algorithms are efficient or inefficient, almost always, aside from the plausible arguments that led the authors to choose the heuristics that drive the algorithms. On the other hand, our results do say something about efficiency on general classes of inputs of all sizes and our analytic methods say why. However, the inputs we can make claims about are CNF Boolean formulas whereas the inputs in, say, the VLSI world are formulas based on multi-level logic. If our results are to apply to real world problems we must either redo our analysis using other models or show that converting to CNF Boolean formulas has little effect on probabilistic results. Neither task is easy and investigation of both tasks are among our long range goals.

Work relating SAT to Design Automation problems is being conducted jointly with Kurt Keutzer of Bell Telephone Laboratories, Murray Hill, New Jersey. Our results to date are given in [15] and mentioned in Section 4.

3. Results on Algorithms for SAT

Let I be a CNF Boolean formula and let $H(I)$ be a heuristic function that outputs a variable contained in I. Let a clause be regarded as a set of literals. Call a clause that contains one literal a unit clause. We have analyzed a wide class of algorithms each of which returns "SAT" if the input instance is satisfiable and "UNSAT" if the input instance is not satisfiable. We call this class SRB (for Search Rearrangement Backtracking) and express it as follows:

SRB(I):
  If I has a null clause then return "UNSAT"
  Else if I is empty then return "SAT"
  Else
    $v \leftarrow H(I)$
    $I_1 \leftarrow \{c - \{\bar{v}\} : c \in I, v \notin c\}$
    $I_2 \leftarrow \{c - \{v\} : c \in I, \bar{v} \notin c\}$
    If SRB(I_1)="UNSAT" and SRB(I_2)="UNSAT" then return "UNSAT"
    Else return "SAT"

In SRB, I_1 is the subinstance of SAT obtained from I by assigning the value *true* to variable v and I_2 is the subinstance obtained by assigning the value *false* to v.

We have also investigated the following two algorithms (which do not backtrack):

$A_1(I)$:
  Construct a random truth assignment t to the variables of I
  Check whether t satisfies I
  If t satisfies I then return "SAT"
  Else return "GIVE UP"

$A_2(I)$:
  While $I \neq \phi$ and $\forall c \in I,\ c \neq \phi$
    If there is a unit clause $\{u\} \in I$ then $v \leftarrow u$
    Else choose a literal v randomly from L
    $I \leftarrow \{c - \{comp(v)\} : c \in I \text{ and } v \notin c\}$
    $L \leftarrow L - \{v, comp(v)\}$
  If $I = \phi$ then return "SAT"
  Else return "GIVE UP"

The algorithms above can easily be modified to return solutions instead of "SAT". The resulting modifications do not significantly affect the efficiency of those algorithms. We chose the above forms because they are easier to analyze.

Here are the results.

Theorem 1: ([11])

Suppose instances of SAT are generated according to model M_2. Let p and r be functions of n.

a. If $\lim_{n\to\infty} \ln(n)/pr < 1$ then the probability that a random instance has a solution tends to 1 as n tends to infinity.

b. If $\lim_{n\to\infty} \ln(n)/pr = c$, $1 \leq c \leq 2.5$, and $\lim_{n\to\infty} n^{1-c^{-1}}/r = \infty$ then the probability that a random instance has a solution tends to 0 as n tends to infinity.

c. If $\lim_{n\to\infty} \ln(n)/pr > 2.5$ then the probability that a random instance has a solution tends to 0 as n tends to infinity.

Theorem 2: ([11])

Suppose instances of SAT are generated according to model M_2. Let p and r be functions of n.

a. If $\lim_{n\to\infty} \ln(n)/pr < 1$ then the probability that algorithm A_1 finds a solution to a random instance tends to 1 as n tends to infinity.

b. If $\lim_{n\to\infty} \ln(n)/pr = c$, $1 \leq c \leq 2.5$, and $\lim_{n\to\infty} n^{1-c^{-1}}/r^{1-\epsilon} < \infty$, for any $\epsilon > 0$, then the probability that algorithm A_2 finds a solution to a random instance tends to 1 as n tends to infinity.

It can be shown that, with probability tending to 1, all variables appear in fewer than $O(\ln(n))$ clauses of a random instance of SAT. Thus, algorithm A_2 almost always runs in $O(n\ln(n))$ time. From this and Theorems 1 and 2 we can assert that, under M_2, almost all satisfiable instances of SAT can be solved in $O(n\ln(n))$ time.

Theorem 3:

Suppose instances of SAT are generated according to model M_1. Let r be a function of n.

a. If $n/r > -1/\lg(1-2^{-k})$ then a random instance of SAT is unsatisfiable with probability tending to 1.

b. If $n/r < -1/\lg(1-2^{-k})$ then the average number of solutions per random instance is exponential in n.

Theorem 3 is analogous to Theorem 1 (for model M_2). The point at which instances change from being mostly unsatisfiable to having a large average number of satisfying assignments is given by $n/r = -1/\lg(1-2^{-k})$ and is called the *flip point*. We use M_1 in place of M_2 when considering the problem of verifying that unsatisfiable instances have no solution because model M_2 generates too many instances with null clauses. If a null clause appears in an instance then that instance is trivially unsatisfiable. Model M_1 does not allow trivial instances of this kind.

Theorem 4: ([14])

Suppose instances of SAT are generated according to model M_1. Let r be a function of n. Then, for all functions H, SRB requires superpolynomial time with probability tending to 1 if $\lim_{n\to\infty} n/r = o(n^{1/\ln\ln(n)})$ and $n/r > -1/\lg(1-2^{-k})$.

Theorems 3 and 4 say that even the most clever heuristic function imaginable cannot give us a probabilistically efficient Backtrack-based algorithm for verifying unsatisfiability if $n/r = o(n^{1/\ln\ln(n)})$ and $n/r > -1/\lg(1-2^{-k})$.

We also have some average case results based on model M_2. Such results give perspective to several average-case papers (e.g. [3], [25], and [26]) by showing the dependence of the favorable results on the presence of null clauses in random instances.

The algorithms below depend on the following definitions. Let a variable which appears exactly once in an instance I be called a *unit* variable. Let a variable which appears exactly twice in I be called a *double* variable. Let a variables which appears at least two times in I be called a *weak-serious* variables. Let a variable which appears at least three times in I be called a *serious* variable. The table below defines substitutions for clauses in I containing unit and double variables. In the table we use v to denote a positive literal taken from a unit or double variable, $\bar{v}$ a negative literal so taken, and x and y either a positive or negative literal which is not necessarily taken from a unit or double variable.

var type	substitution name	occurrence	replacement
unit	unit elimination	$(v, x, ...)$	*true*
unit	unit elimination	$(\bar{v}, x, ...)$	*true*
double	double elimination	$(v, v, x, ...)$	*true*
double	double elimination	$(\bar{v}, \bar{v}, x, ...)$	*true*
double	trivial elimination	$(v, \bar{v}, x, ...)$	*true*
double	pure literal rule	$(v, x, ...), (v, y, ...)$	*true*
double	pure literal rule	$(\bar{v}, x, ...), (\bar{v}, y, ...)$	*true*
double	resolution	$(v, x, ...), (\bar{v}, y, ...)$	$(x, ..., y, ...)$

When we say *apply unit elimination* we mean, according to the table above, look for a clause containing a unit variable v and replace it with the logical value *true*; if no such clause exists do nothing. Similar statements hold for applying any of the other substitution rules listed in the table. It is possible that, after repeated applications of double-variable substitution rules, some double variables will occur only once in I. By *clean up double variables* we mean eliminate all clauses containing double variables that appear once in I.

We consider the following two algorithms:

NULL(I) :

If I has a null clause then return "unsatisfiable"
Otherwise,
Repeatedly apply unit elimination Until opportunities vanish

For all truth assignments t to weak-serious variables in I,
if t satisfies I then return "satisfiable"
Return "unsatisfiable"

INFREQ(I) :

If I has a null clause then return "unsatisfiable"
Otherwise,
Repeatedly apply double variable substitution rules in order
Until opportunities vanish
Clean up all remaining double variables
Repeatedly apply unit elimination Until opportunities vanish
For all truth assignments t to serious variables in I,
if t satisfies I then return "satisfiable"
Return "unsatisfiable"

Theorem 5: ([13])

NULL runs in polynomial average time if

a. $n = r^{\epsilon}$, $.5 > \epsilon > 0$, and $pr \leq r^{.5-\epsilon}$.

b. $n = r^{\epsilon}$, $1 > \epsilon \geq .5$, and $pr < (1-\epsilon)\ln(n)/(2\epsilon)$.

c. $n = \beta r$, $\beta > 0$, and $2.64(1-(1-p)^{2\beta r}(1+2\beta pr)) < \beta e^{-2pr}$.

d. $n = r^{\gamma}$, $\gamma > 1$, and $pr < (\gamma-1)\ln(n)/(2\gamma)$.

The result of Theorem 5b is due strictly to null clauses in random instances yet no other analysis shows a similar result under the same conditions and the result of [3] actually shows that a relatively sophisticated algorithm based on the pure literal rule requires superpolynomial average time under those conditions. The results of Theorem 5a, and 5d match previous results in [25], and [26].

Theorem 6: ([12])

INFREQ runs in polynomial average time if

a. $n = r^{\epsilon}$, $.66 > \epsilon > 0$, and $pr \leq r^{.66-\epsilon}$.

b. $n = r^{\epsilon}$, $1 > \epsilon \geq .66$, and $pr < (1-\epsilon)\ln(n)$.

Both Theorem 6a, and 6b are improvements over the best known previous results under the conditions stated. Theorem 6b is due to the presence of null clauses but Theorem 6a is due to pre-processing the input by eliminating from it infrequently occurring variables (those which are unit or double variables).

Other favorable results have been obtained under model M_1 when instances are nearly always satisfiable. From Theorem 3b this is roughly when $n/r < -1/\lg(1-2^{-k})$. In these studies k is assumed to be independent of n and r. Thus it appears that the case where $\lim_{n,r\to\infty} n/r = \alpha$, where α is any constant greater than zero, is particularly important when considering model M_1.

A number of algorithms have been analyzed under M_1. In [6] we showed that

Theorem 7:

A_2 efficiently solves SAT in bounded probability under M_1 when

$$\lim_{n,r\to\infty} n/r < \frac{2^{k-1}}{k}\left(\frac{k-1}{k-2}\right)^{k-2}.$$

Notice that the expression on the right side of the inequality is $-O(1/k)/\ln(1-2^{-k})$ if k is large.

Theorem 7 is significant for two reasons. First, we cannot make the claim that A_2 almost always finds a solution to a random instance of SAT when one exists, as we could in the case of the random clause model, since there is a large gap between the point where $n/r = -O(1)/\ln(1-2^{-k})$ and the point where A_2 begins to work well probabilistically ($n/r = -O(1/k)/\ln(1-2^{-k})$) due to the $1/k$ factor which appears in the latter term. Second, for that range of n/r over which A_2 is probabilistically efficient, it is only able to find solutions efficiently with bounded probability whereas A_2 finds solutions efficiently in probability under M_2. Thus, we see that, in some sense, model M_1 generates harder instances than M_2 (at least as far as A_1 is concerned) and the results based on the latter model do not map precisely to the same kind of results based on the former.

We also studied the following generalization of A_2:

$A_3(I)$:
    Repeat
        Let c be a smallest clause in I
        Choose u randomly from c
        Remove from I all clauses containing u
        Remove from I all occurrences of $comp(u)$
    Until I is empty or there exist two complementary unit clauses in I
    If I is empty Then return ("satisfiable")
    Otherwise return ("give up")

In [6] we showed

Theorem 8:

A_3 efficiently solves SAT in bounded probability under M_1 when

$$\lim_{n,r\to\infty} n/r < \frac{1.54 * 2^{k-1}}{k+1}\left(\frac{k-1}{k-2}\right)^{k-2} \qquad \text{for } 4 \le k \le 40$$

Theorem 9:

A_3 efficiently solves SAT in probability under M_1 when

$$\lim_{n,r\to\infty} n/r < \frac{0.92 * 2^{k-1}}{k}\left(\frac{k-1}{k-2}\right)^{k-2} \qquad \text{for } 4 \le k \le 40.$$

These results are significant for three reasons. First, A_3 efficiently solves SAT in probability (almost always) over about the same range of n/r that A_2 efficiently solves SAT in bounded probability. Second, the range of n/r over which A_3 is probabilistically efficient is only slightly greater than the range of n/r over which A_2 is probabilistically efficient. Thus, although A_3 performs better than A_2 probabilistically, there is still a wide gap between the flip point and the point at which A_2 begins to perform well. Third, and most important, A_3 and A_2 are vastly superior in probabilistic performance compared to algorithms that rely on certain greedy heuristics to select the next variable to assign a value to. An example of a greedy heuristic is "select the variable v for which the difference between the number of occurrences of the literal v and the literal v' in I is greatest and assign variable v the value which satisfies most clauses". However, as we will see below, greedy heuristics added to A_2 and A_3 improve the performance of those algorithms significantly, especially for the case $k = 3$.

In the case $k = 3$ (CNF expressions with three literals per clause are instances of the 3-Satisfiability problem which is also NP-complete) we have found that the maximum occurring literal selection heuristic (if there are no single-literal clauses in I, select a variable randomly and assign it the value which satisfies most clauses) used with A_2 efficiently solves SAT in bounded probability under M_1 when $\lim_{n,r\to\infty} n/r < 2.9$. In the case $k = 3$, A_2 efficiently solves SAT in bounded probability when $\lim_{n,r\to\infty} n/r < 2.66$ [7]. This may be compared with the flip point ($n/r = 4$).

From our analysis in [6] and [7] we have devised the following algorithm for SAT:

$A_4(I)$:

Repeat

If there is a single-literal clause $\{l\}$ in I Then $u \leftarrow l$

Otherwise $u \leftarrow l^*$ such that $l^* \in L$ and for all $l \in L$ $w(l^*) \geq w(l)$

Remove from I all clauses containing u

Remove from I all occurrences of $comp(u)$

$L \leftarrow L - \{u, comp(u)\}$

Until I is empty or there exist two complementary Unit Clauses in I

If I is empty Then return ("satisfiable")

Otherwise return ("give up")

where $w(l)$, the weight of literal l, is determined as follows:

Let c be a clause in I and let $\mu_j(c)$ be a weighting function mapping clauses to integers. Let us say that $\mu_j(c)$ is the weight of clause c at the end of the j^{th} iteration of $A_4(I)$.

Initially $\mu_0(c) = 1$ for every clause $c \in I$. The clause weighting function is updated as follows: if l is the literal chosen on the j^{th} iteration, $W_j(l)$ is the total weight of clauses containing l at the start of the j^{th} iteration (these clauses will be removed) and $N_j(l)$ is the number of clauses containing $comp(l)$ at the start of the j^{th} iteration (one literal will be removed from each of these clauses) then $\mu_j(c) = \mu_{j-1}(c) + W_j(l)/N_j(l)$ if c contains $comp(l)$, $\mu_j(c) = 0$ if c contains l and $\mu_j(c) = \mu_{j-1}(c)$ otherwise. The literal weighting function is

$$w(l) = W_j(l)/N_j(l)$$

According to our experiments, A_4 solves SAT efficiently in bounded probability under model M_1 when $\lim_{n,r\to\infty} n/r < 4$.

The significance of this result is that A_4 appears to efficiently solve almost all instances of 3-SAT. We hope to prove this result analytically and devise an extension to A_4 which will provide similar performance for any fixed value of k.

4. Recent Results in Design Automation

The problem of designing a test sequence for stuck-at faults in combinational circuits was stated in Section 2. This problem can be reduced to determining a truth assignment which satisfies a Boolean expression. According to our results, this makes stuck-at testing easy, in the probabilistic sense, if the Boolean expressions are in Conjunctive Normal Form. Generally, however, they are not; in fact the Boolean expressions we wish to solve are usually multi-level. In the case of PLAs, the Boolean expressions are two-level and we have attacked them first.

The Boolean expressions associated with PLAs are irredundant and in Disjunctive Normal Form (DNF). That is, no conjunction is subsumed by the remainder of the DNF expression. We examined an input model M_3 which is the same as M_2 with the connectives *and* and *or* reversed and found that instances generated by M_3 are irredundant with probability tending to 1. Thus, we have used M_3 as a model for the Boolean expressions of PLAs.

The internal nodes that must be checked for stuck-at faults are all at the second level. To bring an intermediate node logic level out to a primary output, the logic level of all other intermediate nodes must be *low*. Finding a test vector to do this is equivalent to finding a truth assignment which satisfies a Boolean expression that is the complement of the original expression minus the conjunction associated with the intermediate node under test. We have shown that finding such a truth assignment is easy in a probabilistic sense [15]. The next step is to extend the results to deeper-level logics.

5. Parallel Algorithms and Quadtree Representations

In addition to our probabilistic results on algorithms for NP-complete problems, we have worked with other researchers to obtain average case results on operations for a class of parallel algorithms and on hashing with lazy deletions. The results on parallel computations are presented here and the results on hashing with lazy deletions are presented in the next section. The results on parallel computations are based on a data structure called a Quadtree.

A Quadtree is a natural and well known data structure for the parallel solution of certain numerical problems by means of recursive decomposition. Quadtrees are described in [30] and [31]. A feature of Quadtrees that makes them interesting is their ability to support recursive processes which have no need to communicate with any processes other than the parent process. Unfortunately, there is some overhead penalty that must be paid in terms of space and access time in order to make use of Quadtrees. Specifically, the two overhead factors we are concerned with are (1) the space required to represent a matrix in Quadtree format, and (2) the time required to access an element of a matrix in Quadtree format. The overhead can be kept low only when matrices are sparse (that is, the percentage of zero elements is close to 100 percent). We have chosen to investigate overhead requirements for algorithms involving permutation matrices such as the Fast Fourier Transform (FFT) since an $n \times n$ permutation matrix has exactly n non-zero entries and is, therefore, sparse in the traditional sense when n is large.

We have found that the average space and time requirements to maintain a Quadtree for random permutation matrices are small. Let n be any power of 2. Number all $n \times n$ permutation matrices arbitrarily but uniquely. Let $S_i(n)$ and $T_i(n)$ denote the space and average access time, respectively, required for permutation matrix i. Let $S(n)$ and $T(n)$ denote the average space and time required over all $n \times n$ permutation matrices (we assume that permutation matrices are uniformly distributed). Our results are as follows:

Theorem 10: ([32])

For any $n \times n$ permutation matrix i,

$$S_i(n) \leq \frac{n \lg(n)}{2} + \frac{4n}{3} - \frac{1}{3}, \text{ and}$$
$$T_i(n) \leq \frac{\lg(n)}{2} + \frac{4}{3} - \frac{1}{3n}.$$

We also showed that

Theorem 11: ([32])

$$\bar{S}(n) = \frac{n \lg(n)}{2} + \frac{7n}{8} - \frac{4}{3} \pm \frac{n}{8}, \text{ and}$$
$$\bar{T}(n) = \frac{\lg(n)}{2} + \frac{7}{8} \pm \frac{1}{8}.$$

Furthermore, we showed that overhead requirements for the FFT permutation match the upper bounds of Theorem 10. Theorems 10 and 11 say that Quadtree maintenance overhead results in a modest slowdown factor of $\lg(n)/2$ in both space and time. These costs may be easily recoverable due to the facility for process decomposition and scheduling that is unavailable with other representations.

This work was done jointly with David S. Wise using funds supplied in part by the National Science Foundation under grant number DCR 84-05241.

6. Average Case Analysis of Hashing with Lazy Deletions

A hash table is a collection of nodes, some of which are occupied by cells containing useful data and the rest are unoccupied. For convenience, we assume n nodes numbered arbitrarily from 1 to n. Occasionally cells are "accessed" for the data they contain, or new cells are "inserted" into the table, or existing cells are "deleted" from the table. Let each insertion, deletion, and access be called an *ida* epoch. Suppose a cell c is created at *ida* epoch t_s and deleted at *ida* epoch t_f. The interval (t_s, t_f) is said to be the lifespan of c. At t_s, c becomes the end of a chain of cells (occupied nodes) determined by the hash function employed and the current state of the hash table. Let *chain*(c) denote the chain associated with cell c. During its lifespan, c may be accessed 0, 1, 2 or more times. The number of times that c is accessed during its lifespan is called the accesspan of c. An access of c involves a visit to each of the cells in *chain*(c) up to c. We distinguish between a visit to c and an access of c as follows: c is accessed for data, c is visited for a pointer to the next cell in *chain*(c) or for its data (thus all accesses are also visits). The number of times cells in *chain*(c) are visited when searching for c (for access or deletion) over the lifespan of c is called the searchspan of c. If *chain*(c) does not change during the lifespan of c, as is the case for traditional hashing, then the searchspan of c is the length of *chain*(c) at t_s times the number of searches for c over its lifespan.

The searchspan of c can be reduced by reducing the length of *chain*(c) dynamically. One way to do this is to move c to a node occupied by a cell in *chain*(c) whenever such a cell is deleted (and the node becomes unoccupied). However, this approach, although successful at reducing the searchspan of c, suffers from the high overhead required to make dynamic adjustments to chains every time a cell is deleted. Another approach, with less overhead, is to move c toward the front of *chain*(c) only when c is searched. This is called hashing with lazy deletions. Specifically, hashing with lazy deletions is the result of using the following algorithm to access any cell c:

```
ACCESS(c):
    t ← 0
    i ← 1
    repeat
        d ← hash^i(c)
        if c occupies node d and t > 0 then do the following:
            "access" cell c
            move c to node t
        otherwise, if c occupies node d then
            "access" cell c
        otherwise, if d is unoccupied and t = 0 then
            t ← d
        i ← i + 1
    until c is accessed
```

where $hash^i(c)$ is the i^{th} hash function applied to cell c and the output of $hash^i(c)$, for any $i \geq 1$, is a node (number from 1 to n) in the hash table.

Let $S(c)$ ($A(c)$) denote the searchspan (accesspan) of c and let $\bar{S}$ ($\bar{A}$) be the expectation of $S(c)$ ($A(c)$) over c (respectively). We wish to find $\bar{S}$, $\bar{A}$, and $\bar{S}/(\bar{A}+1)$, the average number of visits per access and deletion of a random cell c.

The model used in the analysis is now described. We consider an arbitrarily long sequence of insertions, deletions, and accesses in the table. The following algorithm, run repeatedly, decides the outcome of each *ida* epoch:

a. Uniformly choose one of n nodes in the table. If we choose an occupied node then do step *b*, otherwise do step *c*.

b. With probability p_d, delete the cell occupying the chosen node (the node becomes unoccupied); otherwise (with probability $1 - p_d$) access the cell occupying the chosen node.

c. With probability p_i, insert a cell at the chosen node (the node becomes occupied); otherwise (with probability $1 - p_i$) do nothing (no *ida* epoch on this go-around).

We choose p_i and p_d such that, over a long sequence of *ida* epochs, the occupancy of nodes in the table reaches "equilibrium" and the average number of occupied nodes in the table is αn. In this case we say the table is α-full. A table is in equilibrium only if the rate at which deletions occur equals the rate at which insertions occur. In our model, if the table is α-full, the rate at which deletions occur is αp_d and the rate at which insertions

occur is $(1-\alpha)p_i$. Thus, we have

$$p_d = \frac{1-\alpha}{\alpha} p_i.$$

It is easy to obtain

Theorem 12: ([4])

$$\bar{A} = (1-p_d)/p_d$$

Our main result is

Theorem 13: ([4])

$$\bar{S} = \sum_{x=1}^{\infty} \frac{(\alpha^{x-1}(1+1/O(n^{1/4}))^{x-1} + 1/O(e^{\alpha\sqrt{n}}))pr(s_1 \geq x | s_0 \geq x)}{1-(1-p_d)pr(s_1 \geq x | s_0 \geq x, \bar{X}_1)}$$

where

$$pr(s_1 \geq x | s_0 \geq x) = \sum_{i=0}^{x-1} \binom{x-1}{i} \alpha^{x-i-1}(1-\alpha)^i \frac{\left(1 - \frac{(1-p_d)pr(Q_{ou,m})}{(1-\alpha)n} - \frac{p_d}{(1-\alpha)n}\right)^i}{1 + \frac{i(p_d+(1-p_d)pr(Q_{ou,m}))}{1-\alpha} + O(\frac{1}{n})}$$

and $Q_{ou,m}$ is the event that there is an unoccupied node in $chain(c_m)$ on the next *idle* epoch given that node m is accessed next.

Although this theorem looks formidable, it is actually quite useful because the sums are not very sensitive to $pr(Q_{ou,m})$. For example, in one extreme case, with $p_d = 1$ ($\bar{A} = 0$), we have

$$\bar{S} = \frac{\bar{S}}{\bar{A}+1} = \sum_{x=1}^{\infty} \alpha^{x-1} = \frac{1}{1-\alpha}.$$

In the other extreme case, with p_d tending to 0 ($\bar{A}$ is large), we have, for large n,

$$\bar{S} \approx \frac{-\ln(1-\alpha)}{\alpha p_d}$$

and

$$\frac{\bar{S}}{\bar{A}+1} \approx \frac{-\ln(1-\alpha)}{\alpha}.$$

If we set $pr(Q_{ou,m}) = 0$ in Theorem 13 we get an easy-to-compute upper bound on $\bar{S}$ for any value of p_d. This upper bound is fairly close to measurements of $\bar{S}$ obtained experimentally. The number of visits per access and deletion of a random cell using Hashing without lazy

deletions is $1/(1-\alpha)$. Thus, the savings in number of visits per access and deletions by using lazy deletions when a table is nearly full and cells are accessed many times before they are deleted is considerable: for example, if $\alpha = .9$ the saving is about 75%, if $\alpha = .95$ the saving is about 85%, and if $\alpha = .99$ the saving is about 96%.

This work was done jointly with Pedro Celis, recent Ph.D. from the University of Waterloo and Assistant Professor of Computer Science at Indiana University.

7. Publications under the grant

1. "Elimination of infrequent variables improves average case performance of Satisfiability algorithms," Technical Report No. 294, Computer Science Department, Indiana University, submitted to *SIAM Journal on Computing.*

2. "On the occurrence of null clauses in random instances of satisfiability," Technical Report No. 291, Computer Science Department, Indiana University, submitted to *Discrete Applied Mathematics.*

3. "Average case analysis of hashing with lazy deletions," with P. Celis, Technical Report No. 260, Computer Science Department, Indiana University, submitted to *Information Sciences.*

4. "Probabilistic analysis of algorithms for stuck-at test generation in PLAs," with Kurt Keutzer, Technical Report No. 278, Computer Science Department, Indiana University (1989).

5. "Search rearrangement backtracking often requires exponential time to verify unsatisfiability," Technical Report No. 210, Computer Science Department, Indiana University, submitted to *SIAM Journal on Computing.*

6. "Costs of quadtree representation of non-dense matrices," with D. S. Wise, Technical Report No. 229, Computer Science Department, Indiana University, to appear in *Journal on Parallel and Distributed Computing.*

7. "Probabilistic analysis of a generalization of the unit-clause literal selection heuristic for the k-satisfiability problem," Technical Report No. 165, Computer Science Department, Indiana University, to appear in *Information Sciences.*

8. "Probabilistic performance of a heuristic for the satisfiability problem," with Y. C. Ho, *Discrete Applied Mathematics* **22** (1988/1989) pp. 35–51.

9. "Correction to probabilistic analysis of the Davis-Putnam Procedure for solving the Satisfiability problem," with J. Plotkin and J. Rosenthal, *Discrete Applied Mathematics* **17** (1987) pp. 295–299.

10. "On the probabilistic performance of algorithms for the Satisfiability problem," *Information Processing Letters* **23** (1986) pp. 103–106.

11. "Probabilistic analysis of two heuristics for the 3-Satisfiability problem," with M. T. Chao, *SIAM Journal on Computing* **15**, No. 4 (1986) pp. 1106–1118.

12. "An approximation algorithm for the maximum independent set problem in cubic planar graphs," with E. Choukhmane, *Networks* **16** (1986) pp. 349–356.

13. "Sensitivity of probabilistic results on algorithms for NP-complete problems to input distribution," *SIGACT News* **17**, No. 1 (1985) pp. 40–59.

14. "Probabilistic analysis of the pure literal heuristic," *Annals of Operations Research* **1** (1984) pp. 273–289.

8. Recent Invited Talks

"Probability in Proof Theory", 14th Symposium on Operations Research, Ulm, Germany (September 8, 1989).

"Analysis of Algorithms for Satisfiability Problems", Workshop on Boolean Functions, Propositional Logic, and AI Systems, at the Research Institute for Applied Knowledge Processing, Ulm, Germany (September 4, 1989).

"Lectures in Scheme: Object Oriented Programming", Research Institute for Applied Knowledge Processing, Ulm, Germany (August 25, 1989).

"Lectures in Scheme: Extend-Syntax", Research Institute for Applied Knowledge Processing, Ulm, Germany (August 18, 1989).

"Lectures in Scheme: Continuations and Call/cc", Research Institute for Applied Knowledge Processing, Ulm, Germany (August 11, 1989).

"Probability in Proof Theory" at The Department of Statistics, University of Rome, Rome, Italy (July 21, 1989).

"Probability in Proof Theory" at The Department of Computer Science, University of Milan, Milan, Italy (July 17, 1989).

"Probability in Proof Theory" at The Department of Computer Science, Universitat Dortmund, Dortmund, Germany (July 11, 1989).

"An Overview of the Scheme Programming Language", Research Institute for Applied Knowledge Processing, Ulm, Germany (July 7, 1989).

"Probability in Proof Theory" at *The Seminar for Natural Language Processing at University of Tubingen*, Tubingen, Germany (June 30, 1989).

"Probabilistic Analysis of Algorithms for VLSI Testing and Design", 1989 CORS/TIMS/ORSA meeting, Vancouver, Canada (May, 1989).

"Probabilistic Analysis of Algorithms for the Satisfiability Problem," at Rutgers University, New Brunswick, New Jersey (January 1989).

"Probabilistic Analysis of Algorithms for the Satisfiability Problem," at the *Workshop on Mathematical Methods in Artificial Intelligence*, Ulm, West Germany (December 1988).

"Probabilistic Analysis of Algorithms for CNF Satisfiability," at ATT Bell Laboratories, Murray Hill, New Jersey (May 1988).

9. Recent Professional Activities

Invited visit to the FAW (Research Institute for Applied Knowledge Processing), Ulm, W. Germany, Summer of 1989.

Workshop organizer, *Workshop on Boolean Functions, Propositional Logic and AI Systems*, Ulm, W. Germany, September, 1989.

Guest Editor: special issue of *Discrete Applied Mathematics* devoted to probabilistic aspects of connections between logic and combinatorics. Targeted for appearance in 1990.

Session chair: CORS/TIMS/ORSA meeting of 1989, May 8-10, Vancouver, Canada. Session title is "Probabilistic aspects of Boolean Functions in Operations Research."

Reviewer for *Journal of the Association for Computing Machinery, SIAM Jornal on Computing, Information Sciences,Annals of Mathematics and Artificial Intelligence, Discrete Applied Mathematics, Mathematical Programming, IEEE Transactions on Computer Aided Design, Annals of Discrete Math, Combinatorics and Complexity, Artificial Intelligence, Air Force Office of Scientific Research.*

10. References

1. Angluin, D. and Valiant, L., "Fast probabilistic algorithms for Hamiltonian Circuits and Matchings," *Proc. 9th Annual ACM Symposium on Theory of Computing* (1977) pp. 30–41.

2. Berman, L. and Hartmanis, J., "On isomorphisms and density of NP and other complete sets," *SIAM J. Comput.* 6 (1977).

3. Bugrara, K., Pan, Y., and Purdom, P. W., "Exponential average time for the pure literal rule," *SIAM J. Comput.* 18 (1989) pp. 409-418.

4. Celis, P. and Franco, J., "Average case analysis of Hashing with lazy deletions," to appear in *Information Sciences.*

5. Chao, M. T., "Probabilistic Analysis and Performance Measurement of Algorithms for the Satisfiability Problem," Ph.D. thesis, Case Western Reserve University, Cleveland, Ohio (1984).

6. Chao, M. T. and Franco, J., "Probabilistic analysis of a generalization of the unit clause literal selection heuristic for the k-Satisfiability problem," to appear in *Information Sciences.*

7. Chao, M. T. and Franco, J., "Probabilistic analysis of two heuristics for the 3-Satisfiability problem," *SIAM J. Comput.*15 (1986), pp. 1106–1118.

8. Chvatal, V., "Determining the stability number of a graph," *SIAM J. Comput.* 6 (1977), pp. 643–662.

9. Coffman, E. G., Frederickson, G. N. and Lueker, G. S., "Probabilistic analysis of the LPT processor scheduling heuristic," in *Deterministic and Stochastic Scheduling*, D. Reidel (1982), pp. 319–331.

10. Davis, M. and Putnam, H., "A computing procedure for quantification theory," *J.ACM* 7 (1960), pp. 201–215.

11. Franco, J., "On the probabilistic performance of algorithms for the Satisfiability problem," *Information Processing Letters* 23 (1986), pp. 103-106.

12. Franco, J., "Elimination of infrequent variables improves average case performance of Satisfiability algorithms," Technical Report No. 294, Computer Science Department, Indiana University (1989), submitted to *SIAM Journal on Computing.*

13. Franco, J., "On the occurrence of null clauses in random instances of Satisfiability," Technical Report No. 291, Computer Science Department, Indiana University (1989), submitted to *Discrete Applied Mathematics.*

14. Franco, J., "Search rearrangement backtracking often requires exponential time to verify unsatisfiability," Technical Report No. 210, Computer Science Department, Indiana University, submitted to *SIAM Journal on Computing.*

15. Franco, J. and Keutzer, K., "Probabilistic analysis of algorithms for stuck-at test generation in PLAs," Technical Report No. 278, Computer Science Department, Indiana University (1989).

16. Franco, J. and Paull, M., "Probabilistic analysis of the Davis Putnam Procedure for solving the Satisfiability problem," *Discrete Applied Math.* 5 (1983), pp. 77-87.

17. Franco, J. and Ho, Y. C., "Probabilistic analysis of a heuristic for the Satisfiability problem," *Discrete Applied Mathematics* 22, (1988/1989), pp. 35–51.

18. Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman (1979).

19. Gill, J., "Computational complexity of probabilistic Turing machines," *SIAM J. Comput.* 6 (1977).

20. Goel, P., "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. on Comput.* c-30 (1981), pp. 215–222.

21. Karmarkar, N., "Probabilistic analysis of some Bin-Packing problems," *Proc.* 23rd *Annual IEEE Symposium on Foundations of Computer Science* (1982), pp. 107-111.

22. Karp, R. M., "Probabilistic analysis of partitioning algorithms for the Traveling Salesman problem in the plane," *Math. Oper. Res.* 2 (1977), pp. 209–224.

23. Karp, R. M., "Probabilistic analysis of some combinatorial search problems," In: J. F. Traub (ed.), *Algorithms and Complexity: New Directions and Recent Results*, Academic Press (1976).

24. Ma, T., Devadas, S., and Sangiovanni-Vincentelli, "Logic verification algorithms and their parallel implementation," *Proc. 24th ACM/IEEE Design Automation Conference* (1987), pp. 283–290.

25. Purdom, P. W., and Brown, C. A., "The pure literal rule and polynomial average time," *SIAM J. Comput.* 14 (1985) pp. 943-953.

26. Purdom, P. W., and Brown, C. A., "Polynomial average time Satisfiability problems," *Information Sciences* 41 (1987) pp. 23-42.

27. Rabin, M. O., "Probabilistic algorithms," In: J. F. Traub (ed.), *Algorithms and Complexity: New Directions and Recent Results*, Academic Press (1976).

28. Schwartz, J. T., "Fast probabilistic algorithms for verification of polynomial identities," *J.ACM* 27 (1980).

29. Wei, R. and Sangiovanni-Vincentelli, "PROTEUS: a logic verification system for combinational circuits," *Proc. 1986 International Test Conference.*

30. Wise, D. S., "Representing matrices as quadtrees for parallel processors," *Information Processing Letters* 20 (1985), pp. 195–199.

31. Wise, D. S., "Parallel decomposition of matrix inversion using quadtrees," *Proc. 1986 International Conference on Parallel Processing* (IEEE Cat. No. 86CH2355-6), pp. 92–99.

32. Wise, D. S. and Franco, J., "Costs of Quadtree representation of non-dense matrices," to appear in *Journal of Parallel and Distributed Computing.*